

**SYBEX Advanced Viewing Sample Chapter**

# **.NET Framework Solutions: In Search of the Lost Win32 API**

**John Paul Mueller**

## **Advanced Viewing Sample Chapter Chapter 4: Processing Windows Messages**

Copyright © 2002 SYBEX Inc., 1151 Marina Village Parkway, Alameda, CA 94501. World rights reserved. No part of this publication may be stored in a retrieval system, transmitted, or reproduced in any way, including but not limited to photocopy, photograph, magnetic or other record, without the prior agreement and written permission of the publisher.

ISBN: 0-7821-4134-X

SYBEX and the SYBEX logo are either registered trademarks or trademarks of SYBEX Inc. in the USA and other countries.

TRADEMARKS: Sybex has attempted throughout this book to distinguish proprietary trademarks from descriptive terms by following the capitalization style used by the manufacturer. Copyrights and trademarks of all products and services listed or described herein are property of their respective owners and companies. All rules and laws pertaining to said copyrights and trademarks are inferred.

This document may contain images, text, trademarks, logos, and/or other material owned by third parties. All rights reserved. Such material may not be copied, distributed, transmitted, or stored without the express, prior, written consent of the owner.

The author and publisher have made their best efforts to prepare this book, and the content is based upon final release software whenever possible. Portions of the manuscript may be based upon pre-release versions supplied by software manufacturers. The author and the publisher make no representation or warranties of any kind with regard to the completeness or accuracy of the contents herein and accept no liability of any kind including but not limited to performance, merchantability, fitness for any particular purpose, or any losses or damages of any kind caused or alleged to be caused directly or indirectly from this book.

Sybex Inc.  
1151 Marina Village Parkway  
Alameda, CA 94501  
U.S.A.  
Phone: 510-523-8233  
[www.sybex.com](http://www.sybex.com)




## CHAPTER 4

---

# Processing Windows Messages

---

- Understanding the Windows Message Types
  - Windows Message Handlers Found in the .NET Framework
  - An Overview of Windows Message Functions
  - Creating a Windows Message Handler Example
- 

**A**s part of its effort to hide some of the mundane details of how Windows works from developers, Microsoft has actually hidden a few too many facts. One of the issues you need to know about is how Windows uses and processes messages. For that matter, you need to know how to create your own messages at times. While the built-in message handling provided by Microsoft for the .NET Framework environment works most of the time, there are a few situations where you might want to have more control than the environment provides.

This chapter discusses all of the aspects of Windows message processing, handling, and generation. You'll learn about the message pump and some of the other low-level details that the .NET Framework normally hides. In addition, we'll look at some of the messages that the .NET Framework handles for you. For example, you have access to the `WM_HELP` message—it's just hidden by a form event handler (more on this topic in the "Windows Message Handlers Found in the .NET Framework" section of the chapter).

The important bit of information to get from this chapter is that Windows uses a messaging system to communicate with applications that's remained essentially unchanged from the days of Windows 3.x. The applications can also communicate with Windows and other applications using messages. In short, understanding the messaging system is essential if you want to build robust applications.

## Understanding the Windows Message Types

Windows is literally packed with messages. There are thousands of messages to which your application can respond. Even the user interface messages number over a thousand. There are messages for mouse movement, keyboard clicks, system messages, user messages, all kinds of messages. One of the best places to view these messages in action is Spy++. Figure 4.1 shows the Messages tab of the Message Options dialog box. As you can see, there are a number of ways to group just the messages that appear in the user interface portion of Windows. (You can learn more about Spy++ in the "Spy++" section of Chapter 4—messages appear in the "Viewing Messages" subsection.)

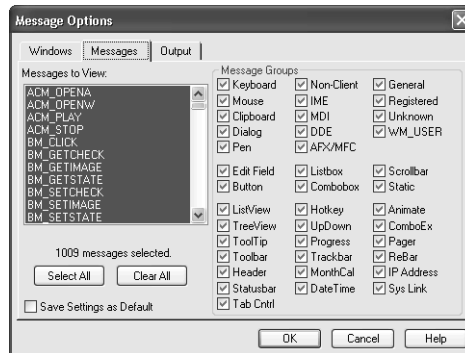
User interface messages don't exist in a vacuum—you aren't going to find many messages that exist in isolation. Consider the simple act of clicking a button. When a user clicks the button, it generates a `WM_LBUTTONDOWN` message, and then a `WM_LBUTTONUP` message. If the user presses and releases the mouse button within the time limits of a click, the act also generates a `BN_CLICKED` message, which Windows sends to the button's parent. This action generates a `BM_CLICK` message, which is where event handlers normally act on the button click (both managed and unmanaged).

However, the presence of a `WM_LBUTTONDOWN` message doesn't necessarily signify a button click—it also occurs for radio button selections and other control events, so Windows has to

know which control is the focus of the action. In addition, the `WM_LBUTTONDOWN` message can begin a drag-and-drop action. The user might not want to actually select the item; they may simply want to move it. (Of course, dragging and dropping one or more objects normally entails some form of selection—it may simply mean the object isn’t activated.)

**FIGURE 4.1:**

Spy++ provides one of the best ways to graphically see the effects of messages.



Something I haven’t covered in all of this is the appearance of ancillary messages when a user performs most tasks. For example, looking at the same button click from the previous paragraph, the user doesn’t just click a button; the mouse cursor has to appear in the right place to perform the click. This means generating a wealth of `WM_MOUSEMOVE` messages as the user moves the mouse to the correct position in the dialog box. If the user moves the mouse outside the dialog area, then the dialog will also receive a `WM_MOUSELEAVE` message. There’s no `WM_MOUSEENTER` message because the `WM_MOUSEMOVE` message serves to tell the application of the mouse entry into the dialog area.

Fortunately, even in unmanaged applications, a developer only has to track the messages of interest and can ignore everything else. The developer doesn’t have to track every message and suffer the repercussions of message overload. The .NET Framework further reduces the need to track messages—as we’ll see in the “Windows Message Handlers Found in the .NET Framework” section that follows.

In many cases, a user interface message doesn’t affect the user interface directly—it acts in the background as a notification of change. For example, Windows issues the `WM_SYSCOLOR-CHANGE` message to all top-level windows whenever the user changes the system colors. This notification helps the application maintain the appearance of controls and data so that the reader can still see the screen, no matter how garish the color selection.

There’s a group of messages that affect the application as a whole, including the user interface, but act in the background instead of the foreground. For example, Windows issues a `WM_POWER` message every time the machine is about to enter the suspended mode. An application can track the `WM_POWER` message to determine when the system is about to suspend operations. It

can use this message as a way to determine when to save application status information or open data files.

System-level messages often provide two-way communication for applications. An application can receive a system-level notification such as the WM\_POWER message. It can also issue a system-level request using the WM\_SYSCOMMAND message. In this last case, the command often has nothing to do with the user interface of an application at all, but does affect system or individual application operation. In many cases, the request is for background services that the operating system can provide anonymously in an asynchronous fashion.

If you haven't noticed already, all of the messages we've discussed have a two-letter identifier followed by the message function, task, or type. The two-letter identifier provides the best means of classifying most (but not all) messages. Table 4.1 provides a list of the most common message types with accompanying short description.

**NOTE**

Table 4.1 doesn't contain a complete list of all prefixes. It includes a list of common prefixes that the developer is likely to use or care about. For example, the list doesn't contain the NM prefix because these messages are normally used at a low level for parent control notification of events handled at a higher level by application code.

**TABLE 4.1:** Common Prefixes for Win32 API Messages

Prefix	Description
BCM	A button control message that changes the resource usage or other inner workings of a button or a button-like control.
BCN	A button control notification that notifies the control of a change in status.
BM	A button message that obtains status information about the button or its appearance. This is a general prefix used for all button-like controls.
BN	A button notification that specifies a change in the button status, such as a user click. This is a general prefix used for all button-like controls.
CB	A control box specific status, resource, or setup message. Note that these entries tend to be unique and you'll need to watch button and edit box messages as well.
CBEM	An extended control box specific status, resource, or setup message.
CDM	A common dialog box message that obtains status information about the dialog box or its appearance.
CDN	A common dialog box message that specifies a change in dialog box status, such as the user clicking the OK button.
DBT	A device specific message generally used to signal a device status change, such as a configuration change or when a user plugs in a new device.
DTM	A date/time picker message that obtains status information about the control or its appearance.

*Continued on next page*

**TABLE 4.1 CONTINUED:** Common Prefixes for Win32 API Messages

Prefix	Description
DTN	A date/time picker message that specifies a change in control status, such as a change in the date or time format.
EM	An edit box message that obtains status information about the edit box or its appearance. This is a general prefix used for all edit box-like controls.
EN	An edit box notification that specifies a change in the edit box status, such as a change in the text content. This is a general prefix used for all edit box-like controls.
LB	A list box specific status, resource, or setup message.
LBN	A list box message that specifies a change in list box status, such as the user double-clicking an item. These items tend to be very specialized, so you'll need to watch standard button messages (BN) as well. Depending on the configuration of the list box, you'll want to consider the EN and CB messages as well.
LBS	A list box message that specifies a change in the list box style, such as a change in the technique used to sort list box entries.
LVM	A list view message that obtains status information about the control or its appearance.
LVN	A list view message that specifies a change in control status, such as an item change or selection.
LVS	A list view message that specifies a change in the list view style, such as a change in the technique used to sort list view entries. The styles are a lot more comprehensive than those used for LBS messages. For example, there are separate styles for sorting in ascending and descending order.
MM	A multimedia message—usually hardware or media specific. For example, this group includes messages that affect the joystick. It also includes messages that denote multimedia events such as opening a waveform file.
SC	A system control message such as a request to turn on the screensaver or shut the system down.
TB	A toolbar specific status, resource, or setup message.
TBM	A trackbar specific status, resource, or setup message.
TVM	A tree-view specific status, resource, or setup message.
UDM	An up-down control specific status, resource, or setup message.
WM	A generic Windows message used for a variety of purposes including system requests. In some cases, the message will contain service information. For example, you'll find a series of WM_ADS messages that reflect changes in Active Directory Service status.

Notice that most of these message categories are user interface specific. That's because many of the messages that Windows handles are user-generated—a user does something and generates a message as a result. In addition, some controls have specific status and notification classes (such as BM and BN) while others don't (such as CB). Windows will often classify a control as having button-like behavior. It uses the button-related messages to handle user events and programmatic changes to those controls.

Applications can also create messages, and you'll find custom messages for certain classes of applications. For example, many database managers (DBMSs) use the DB message prefix to signal database events such as a new record. In many cases, such as the DB example, you'll find common application messages documented in the Platform SDK help (but not in the Visual Studio .NET help).

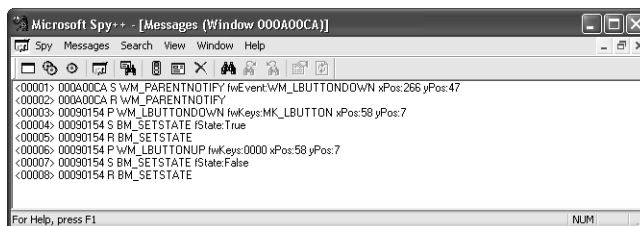
A few Win32 API messages are application specific and you'll generally need to consider them only when using Visual C++ (some can be helpful in other situations). For example, the DT prefix is used for text drawing messages. Most of these messages do appear in the Visual Studio .NET documentation because the current version of Visual C++ .NET uses them.

## Windows Message Handlers Found in the .NET Framework

The previous section might leave you feeling hopelessly mired in messages you don't know about and aren't sure that you want to know about. However, we'll see as the book progresses that you need to know about messages because that's the only form of communication Windows actually recognizes. In addition, Spy++ can be the most valuable tool in your Win32 API toolbox. However, you don't need to memorize all of these messages and you'll find that you don't have to worry about every message that Windows can process. You'll find that the .NET Framework implements the most common messages for you as event handlers. For example, the `Click()` event handler is a response to the `BN_CLICKED` notification. The Spy++ display in Figure 4.2 shows the messages generated for the `ShowMessage` application from Chapter 3 when you click the `Test` button.

**FIGURE 4.2:**

Spy++ can provide clues on which messages you need to implement.

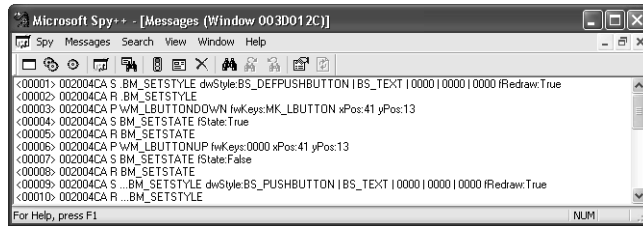


As you can see, the parent window receives a message that the user has clicked the left button (`WM_PARENTNOTIFY` with the `WM_LBUTTONDOWN` message as data). This message is passed to the child window (a button, in this case), which registers the `WM_LBUTTONDOWN` message and takes control of the message stream. The button also sets its state to true. The user releases the left mouse button, which generates a `WM_LBUTTONUP` message and the requisite setting changes for the button. This set of events ends a `BN_CLICKED` event (not shown).

The same messages occur no matter what type of application you create—managed or unmanaged. Figure 4.3 shows an example of a simple unmanaged Visual C++ application (located in the \Chapter 04\SayHello folder of the CD). The MFC application has a few additional bells and whistles, such as the selection of a default button, but otherwise the message sequence is the same.

**FIGURE 4.3:**

Managed and unmanaged applications both generate the same sequence of messages.



**TIP**

Sometimes it's difficult to correlate a .NET Framework event handler with a Win32 API message. In fact, you might not know whether the .NET Framework even provides support for a given message. In many cases, you can create a test application that uses what you suspect is a Win32 API message, and then see if Spy++ reports that the message is active in the test application. If the message isn't active, then you've at least eliminated some of the .NET Framework functionality that could support the message. Because Microsoft hasn't provided any documentation that shows the correlation between .NET Framework event handlers and Win32 API messages, you'll occasionally need to perform this type of interactive research—making Spy++ one of your best friends.

As previously mentioned, Microsoft hasn't created a list that shows the correlation between Win32 API messages and the event handlers found in the .NET Framework, so you need to test as you go along. However, in general, you can count on the .NET Framework handling all general control messages, as well as many system-related messages. For example, you'll find that the .NET Framework handles all of the button messages you'll ever need, so there shouldn't ever be a need to implement a message handler for a button. The same holds true for edit and list boxes.

The .NET Framework doesn't handle some system messages such as turning the screensaver on or off, or changing the display settings. The .NET Framework doesn't implement any of the Windows XP-specific messages. For example, you won't find any support for the Fast User Switching feature. All of these messages will require some type of application support.

You'll also find a lack of low-level device support in the .NET Framework. For example, general printer commands are handled, but anything going to the parallel port (such as LPT1) isn't. Newer devices such as USB (Universal Serial Bus) drives are handled by the operating



system for the most part and you can access the data they contain using standard .NET file handling calls. However, if you need to access the drive itself, you'll need to create special message handlers (as well as a wealth of other device handling code).

## An Overview of Windows Message Functions

Windows messages don't suddenly jump out of your application and appear in a message queue somewhere. You need to generate the messages you want to send to another application using a function such as `SendMessage()`. To receive a message, your application must provide some type of listening mechanism, which is going to be an event in most cases. The application will need to override the standard .NET message pump to generate the event and provide an event handler to perform some task based on the event.

This section of the chapter discusses some of the Win32 API message functions you'll use to send and receive Windows messages. It's important to note that sending messages requires a complete Win32 API call setup, while receiving messages requires hooks into the existing .NET Framework classes. In short, the information below is a starting point—it's the Win32 API portion of the picture. We'll look at how all of the pieces fit within the .NET Framework in the various examples in this chapter.

---

**NOTE**

You might also want to look at the `ScreenSaver` example in Chapter 3, which demonstrates the simplest way to send a message to Windows that results in a system action. The examples in this chapter are a little more complex, so the Chapter 3 example is a good place to start.

### `SendMessage()`

`SendMessage()` is the simplest function you can use to send messages to Windows or to other applications. One of the best ways to test this function is to work with the system commands. You'll find a complete test program for the system commands in the `\Chapter 04\C#\SysCommand` and `\Chapter 04\VB\SysCommand` folders of the CD. This example shows the full set of SC commands in action (at least those that are documented). Figure 4.4 shows the dialog for this example, which includes a list of the SC commands, along with one long string for testing the vertical scroll command (`SC_VSCROLL`).

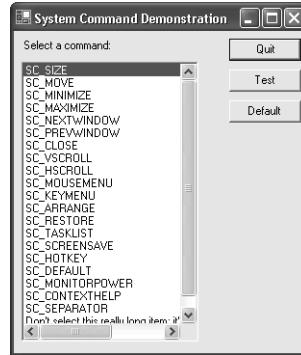
---

**NOTE**

Some commands in the list will only work if you trigger the `Test` button with the `Enter` key, instead of clicking `Test` with the mouse. The reason is that the action takes place immediately—the mouse cursor changes to the double-pointed or other arrow type. Unfortunately, because the mouse is already in use, the command fails. The only way to get around this problem for testing is to use the keyboard in place of the mouse.

**FIGURE 4.4:**

The SysCommand example shows how the various system commands work.



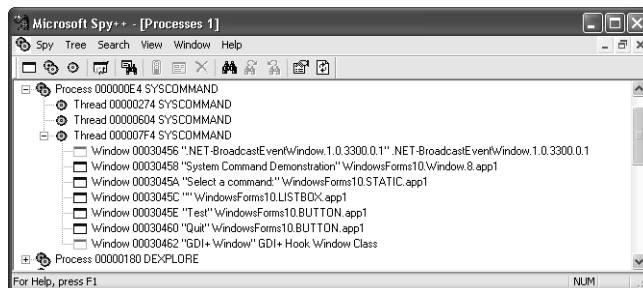
Some of the system commands require special handling. For example, the `SC_MONITORPOWER` command requires input in the `lParam` argument. The standard value of 0 doesn't accomplish anything. If you input 1, then the display will go to a low power state, while a value of 2 turns the display off. The example uses a value of 2 to ensure that most systems will see at least a momentary power down of the screen. In some cases, you might have to modify the display settings to get this system command to work properly. Here's the modified code.

```
DoSysCommand = (Int32)SystemCommand.SC_MONITORPOWER;
SendMessage(this.Handle, WM_SYSCOMMAND, DoSysCommand, 2);
return;
```

Notice that we're still using the handle for the main window. Figure 4.5 shows another view from Spy++. Notice that each of the major controls in the application is also a window. The window-like quality of the controls enables you to access them by sending them messages. Of course, the control has to have some means of responding to the messages—there's no magic involved.

**FIGURE 4.5:**

Every visible control in an application is very likely a window as well.



**WARNING**

Figure 4.5 also shows two hidden windows—those with a grayed outline. The first is for .NET broadcasts, while the second is for GDI+, as opposed to the plain GDI used by standard Windows applications. In most cases, you won't want to modify these windows or send messages to them—you might see unpredictable results.

To send a message to a control, you need to provide both a handle for the control's window and an event handler to listen for the message. In some cases, such as moving a window, the .NET Framework provides a default handler. You can also add your own handler that CLR will call when the .NET portion of the call completes. The following code shows how to obtain the handle for `lbCommandSel` and use it to move the list box around.

```
IntPtr Temp = IntPtr.Zero; // A temporary handle.

// Load the proper system command.
DoSysCommand = (Int32)SystemCommand.SC_MOVE;

// Obtain the handle to the list box.
Temp = lbCommandSel.Handle;

// Move the list box instead of the main window.
SendMessage(Temp, WM_SYSCOMMAND, DoSysCommand, 0);
return;
```

You must highlight the **SC\_MOVE (List Box)** entry in the list, tab twice to highlight the Test button, and then press Enter. Clicking Test will cause the code to fail because the mouse is already engaged in clicking. You'll see the mouse cursor change to a quad-ended move cursor. Moving the mouse will move the list box. Press Enter again and you'll see the ending message found in the `lbCommandSel_Move()` method. Normally, you'd place any code required to end the move command in this method, but the example uses a simple message box.

## PostMessage(), PostThreadMessage, and PostQuitMessage()

The `PostMessage()` and the `SendMessage()` functions perform essentially the same task—they both send a message to the specified thread. However, the `PostMessage()` function returns from the call immediately, while the `SendMessage()` function waits for the recipient to respond. Both functions accept essentially the same arguments, so anything you can do with a `SendMessage()` call, you can do with a `PostMessage()` call.

You might wonder why Microsoft would include two calls with essentially the same functionality. There's a distinct disadvantage when using the `PostMessage()` call—you don't know if anyone received the message. It's best to use `PostMessage()` when you don't care if anyone receives or acts upon the call.

**NOTE**

There are a number of superceded message functions that you'll still find in the Platform SDK help and in the C/C++ header files. Never use these functions within an application because Microsoft doesn't support them and you don't know if they'll work in future versions of Windows. For example, the `PostAppMessage()` function has been replaced by the `PostThreadMessage()` function. Unfortunately, examples of these old functions prevail online and you even see them in the help files. Refer to the "Obsolete Windows Programming Elements" topic in the Platform SDK help file for a list of old functions and their replacements. This list isn't complete, but it's good place to start.

Both the `PostMessage()` and `SendMessage()` functions can accept special handles. However, one of these special handles works better with `PostMessage()` because it doesn't incur the delay that using `SendMessage()` would incur. You can use the `HWND_BROADCAST` handle to tell Windows to send a particular message to every accessible window. For example, you might use such a call to restore all of the windows as shown in Listing 4.1. (The example code appears in the \Chapter 04\C#\MinimizeAll and \Chapter 04\VB\MinimizeAll folders of the CD.)


**Listing 4.1      Using the HWND\_BROADCAST Handle to Call All Windows**

```
// Used to send a system command message.
[DllImport("User32.DLL")]
public static extern int PostMessage(IntPtr hWnd,
                                   UInt32 Msg,
                                   Int32 wParam,
                                   Int32 lParam);

// The WM_SYSCOMMAND constant used to access the SC constants.
public const Int32 WM_SYSCOMMAND = 0x112;

// The HWND_BROADCAST handle sends the message to all windows.
public IntPtr HWND_BROADCAST = new IntPtr(0xFFFF);

// A list of SC constants used for all types of system
// command access.
public enum SystemCommand
{
    SC_SIZE           = 0xF000,
    SC_MOVE           = 0xF010,
    SC_MINIMIZE       = 0xF020,
    SC_MAXIMIZE       = 0xF030,
    SC_NEXTWINDOW     = 0xF040,
    SC_PREVWINDOW     = 0xF050,
    SC_CLOSE          = 0xF060,
    SC_VSCROLL        = 0xF070,
    SC_HSCROLL        = 0xF080,
    SC_MOUSEMENU      = 0xF090,
    SC_KEYMENU        = 0xF100,
```

```

        SC_ARRANGE      = 0xF110,
        SC_RESTORE     = 0xF120,
        SC_TASKLIST    = 0xF130,
        SC_SCREENSAVE  = 0xF140,
        SC_HOTKEY      = 0xF150,
        SC_DEFAULT     = 0xF160,
        SC_MONITORPOWER = 0xF170,
        SC_CONTEXTHELP = 0xF180,
        SC_SEPARATOR   = 0xF00F
    }

    private void btnTest_Click(object sender, System.EventArgs e)
    {
        // Minimize all of the windows.
        PostMessage(HWND_BROADCAST,
                    WM_SYSCOMMAND,
                    (Int32)SystemCommand.SC_RESTORE,
                    0);
    }

```

---

As you can see, using `PostMessage()` with the broadcast handle is essentially the same as using `SendMessage()`—the main difference is that the function returns immediately. If you try using this code with `SendMessage()` in place of `PostMessage()`, you'll see a definite delay as `SendMessage()` waits for all of the windows to return a response.

This code has an interesting side effect. Not only does it restore all of the visible windows, but it restores all of the hidden windows as well. The resulting chaos might look unappealing, but I've actually learned about a few windows that don't appear in the Spy++ list, but do appear on screen after using this call. Log off and back on your machine to restore the screen—a reboot isn't necessary to hide the hidden windows again.

A second special handle accesses the Desktop. The `HWND_DESKTOP` handle enables you to send messages to the Desktop using either `PostMessage()` or `SendMessage()`. Here's the definition for `HWND_DESKTOP`.

```

// The HWND_DESKTOP handle sends message only to the Desktop.
public IntPtr HWND_DESKTOP = new IntPtr(0);

```

The `AddFontFile()` method replicates the functionality of the `AddFontResource()` function of the Win32 API. Both enable you to add private fonts to your application without registering them within Windows first. Either form of the function works fine if you want to share your registered font with every other application running in Windows. However, what happens if you want to keep your special font truly secret? You need to use the `AddFontResourceEx()` function. This Win32 API function includes flags that keep your font secret and prevent other applications from enumerating the font. However, no matter which function you use to load

a font, you still have to tell everyone that there was a change to the font table, which means sending a message. The code in Listing 4.2 shows how to load the `VisualUI.TTF` font that appears on most hard drives with Visual Studio .NET (among other applications) installed.

**NOTE**

For best viewing results, run this example application outside of the debugger. If you load the font while within the debugger, it tends to stay in memory until you exit the Visual Studio IDE. You can still follow code execution within the debugger to see how the various calls work—the only problem is that the font won't unload. That's because Windows associates the font with the Visual Studio IDE instead of the application since the application is executing within the debugger environment.

**Listing 4.2      Use the `AddFontFileEx()` Function to Load Fonts Privately.**

```
// The function required to add a private font resource.
[DllImport("GDI32.DLL")]
public static extern int AddFontResourceEx(String lpszFilename,
                                         Int32 fl,
                                         IntPtr pdv);

// The function required to remove a private font resource.
[DllImport("GDI32.DLL")]
public static extern bool RemoveFontResourceEx(String lpszFilename,
                                              Int32 fl,
                                              IntPtr pdv);

// Flags used to define how the private font resource is loaded.
public const Int32 FR_PRIVATE      = 0x10;
public const Int32 FR_NOT_ENUM    = 0x20;

// Used to send a system command message.
[DllImport("User32.DLL")]
public static extern int PostMessage(IntPtr hWnd,
                                     UInt32 Msg,
                                     Int32 wParam,
                                     Int32 lParam);

// The WM_SYSCOMMAND constant used to access the SC constants.
public const Int32 WM_FONTCHANGE = 0x001D;

// The HWND_BROADCAST handle sends the message to all windows.
public IntPtr HWND_BROADCAST = new IntPtr(0xFFFF);

System.Drawing.Text.PrivateFontCollection PFC;

private void btnLoadFont_Click(object sender, System.EventArgs e)
{
    // Determine which action to take.
```

```

if (btnLoadFont.Text == "Load Font")
{
    int    Result = 0; // Results of loading the font.

    // Load the desired font.
    Result = AddFontResourceEx(
        "D:\\Program Files\\Microsoft Visual Studio
.NET\\Common7\\IDE\\VisualUI.TTF",
        FR_PRIVATE,
        IntPtr.Zero);

    // Check the results.
    if (Result == 0)

        // Display an error message if necessary.
        MessageBox.Show("The font failed to load for some reason.",
            "Load Failure",
            MessageBoxButtons.OK,
            MessageBoxIcon.Error);
    else
    {
        // Change the button caption.
        btnLoadFont.Text = "Unload Font";

        // Tell everyone we've loaded a new font.
        PostMessage(HWND_BROADCAST, WM_FONTCHANGE, 0, 0);
    }
}
else
{
    bool    Result; // Results of loading the font.

    // Load the desired font.
    Result = RemoveFontResourceEx(
        "D:\\Program Files\\Microsoft Visual Studio
.NET\\Common7\\IDE\\VisualUI.TTF",
        FR_PRIVATE,
        IntPtr.Zero);

    // Check the results.
    if (!Result)

        // Display an error message if necessary.
        MessageBox.Show("The font failed to unload for some reason.",
            "Unload Failure",
            MessageBoxButtons.OK,
            MessageBoxIcon.Error);
    else
    {
        // Change the button caption.
        btnLoadFont.Text = "Load Font";
    }
}

```

```

        // Tell everyone we've loaded a new font.
        PostMessage(HWND_BROADCAST, WM_FONTCHANGE, 0, 0);
    }
}

private void btnDisplayDialog_Click(object sender, System.EventArgs e)
{
    // Display the font dialog.
    dlgFont.ShowDialog(this);
}

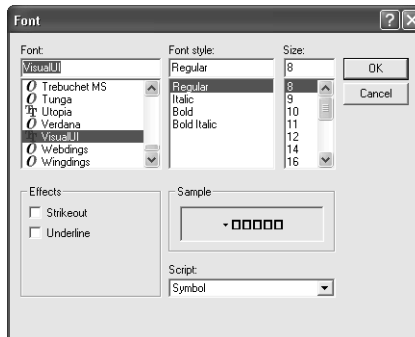
```

As you can see, the sample code can load and unload the VisualUI.TTF font. The `AddFontResourceEx()` and `RemoveFontResourceEx()` function calls load the font publicly if you don't specify any flags or privately if you specify the `FR_PRIVATE` flag shown. Notice the use of `PostMessage()` in this example. You must tell other windows about the new font or they won't recognize it (this includes other windows in the current application). The `WM_FONTCHANGE` message doesn't require any parameters—the other windows will create a fresh enumeration of the font list if necessary.

If you click Display Fonts immediately after loading the example application, you'll notice that the VisualUI is missing from the list. Load the font with the code shown in Listing 4.2 and you'll see the VisualUI font in the list as shown in Figure 4.6.

**FIGURE 4.6:**

Loading the VisualUI font using the default code displays it in the Font dialog box.



There are some interesting changes you can make to the code in Listing 4.2. For example, try the example with a `SendMessage()` in place of a `PostMessage()` call and you'll see the time differential can be significant. Try running the call without sending the `WM_FONTCHANGE` message at all and you'll notice that not even the local application will notice it in some cases (the change becomes intermittent). Try loading the font publicly (without any flags). Other applications such as Word will contain the font in their font list. Reboot the machine after a public



load to ensure that the font is removed from memory. Now, try using the `FR_NOT_ENUM` flag when loading the font and you'll notice that only the test application displays the font.

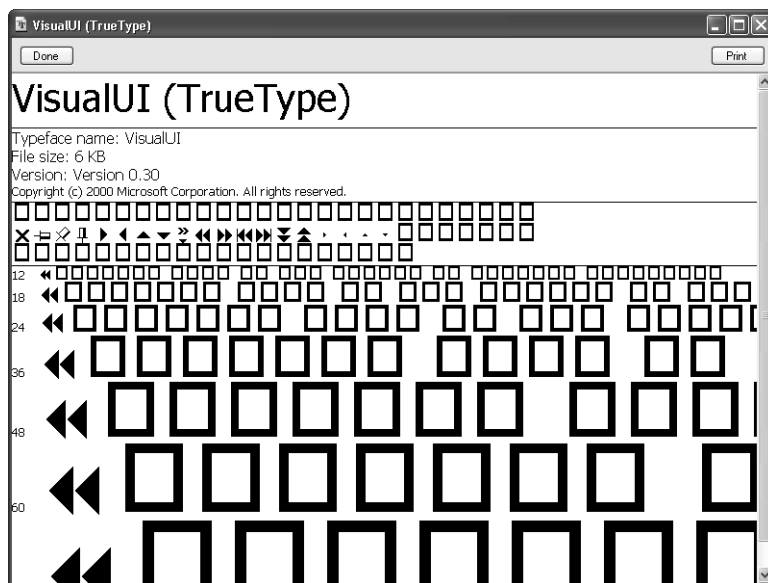
**NOTE**

The `AddFontResourceEx()` function, like many of the special functions in the book, isn't supported by Windows 9x systems, including Windows Me. In addition, the fonts you add using this function are only accessible for the current session—they're unloaded as soon as the user reboots the machine. As you can see, it's essential to check the Platform SDK documentation for limitations on using Win32 API functions directly.

The `VisualUI.TTF` font is interesting for developers, but almost useless for users, so it makes a perfect private font. Figure 4.7 shows what this font looks like. As you can see, it contains the special font Microsoft uses for drawing the VCR-like controls on screen. It also contains some unique graphics such as the pushpin used in some areas of the Visual Studio IDE. Having access to these special graphics can save development time.

**FIGURE 4.7:**

The `VisualUI` font may not have much to offer users, but it can save some drawing time for developers.



There are several variations on the `PostMessage()` function. One of the more interesting messages is `PostThreadMessage()`. This form of the function enables you to post a message to the threads of the current application. You still provide `Msg`, `lParam`, and `wParam` arguments. However, instead of a window handle, you need to provide a thread identifier. The `PostThreadMessage()` function has several constraints, including a special constraint under Windows 2000 and Windows XP—the thread identifier must belong to the same desktop as the calling thread or to a process with the same Locally Unique Identifier (LUID).

You'll almost never need to use the `PostQuitMessage()` function. All .NET languages have a built-in method to perform this task and you're better off using it whenever possible. The `PostQuitMessage()` tells Windows that your application is going to exit. It includes an exit code that an external application can use to determine the exit status of your application (generally 0 for a successful exit). It's important to know about this function because it does come in handy in certain rare circumstances—mainly within wrapper DLLs. You can use this message to force an application exit when catastrophic events occur. The only time you should consider using this message is if the application is hopelessly frozen and you still want to provide some means of exit (so the user doesn't have to perform the task manually). In short, for the .NET developer, this is the message of last resort.

## SendNotifyMessage()

Sometimes you need a message whose behavior depends on the circumstance in which it's used. The `SendNotifyMessage()` function combines aspects of the `SendMessage()` and the `PostMessage()` functions we discussed earlier. When you use `SendNotifyMessage()` to send a message to the window process in the same thread, it waits for a response. On the other hand, if you send the message to a window in another thread, `SendNotifyMessage()` returns immediately. This duality of function ensures you gain reliable message transfer for the local thread, without the performance delay of waiting for other threads to complete their work.

### WARNING

Avoid using pointers in any asynchronous message function, including `SendNotifyMessage()`, `PostMessage()`, `SendMessageCallback()`, because the function will likely fail. The message call will return before the recipient can look at the data pointed at by the pointer, which means the recipient may not have access to the data required to complete the call. For example, the caller could deallocate the memory used by the data immediately upon return from the call. If you need to send a pointer as part of a message, then use the `SendMessage()` function to ensure the recipient is done using the pointer before the message returns. While this technique does incur a performance penalty, it also ensures that the message will complete as anticipated.

The `SendNotifyMessage()` function requires the same input as both `SendMessage()` and `PostMessage()`. You can use it to send both single-window and broadcast messages.

## SendMessageCallback()

The `SendMessageCallback()` function has two main purposes. First, it sends a message to another process—just like the other message-related functions we've discussed so far. Second, it registers a callback function with the message recipient. A callback function is a special function used by the message recipient to return message results to the message sender. In short, this is the first function to provide a two-way communication path for messages.

The first four arguments for the `SendMessageCallback()` function are the same as any other message function. You need to provide an *hWnd*, *msg*, *lParam*, and *wParam* values. The fifth argument, *lpCallback*, is a pointer to a callback function. This requirement means you need to use a delegate to pass the address pointer. We'll see how this works in Chapter 5, which concentrates on callback functions. The sixth argument, *dwData*, is a value that you can pass from your application, through the message recipient, and back to the callback function. This application-defined value can provide input to the callback function that determines how it processes the message return data.

You'll normally use the `SendMessageCallback()` function to retrieve data from a device, system service, or other data source that the .NET framework doesn't support directly. For example, you could use this technique to obtain an enumeration of the devices located on a USB.

## GetMessage() and PeekMessage()

We've discussed the Windows message pump and several of the messages that can appear within the message queue. You know that whenever an application sends a message, Windows will place the message in the recipient's message queue, which is essentially an "In Box" for Windows messages. However, we haven't discussed how the recipient actually receives the message so it can act on it. The `GetMessage()` and the `PeekMessage()` functions provide the means for retrieving a message from the Windows message queue so the application can act on it. Use the `GetMessage()` function to remove the message from the queue and the `PeekMessage()` function to see if the message exists without removing it.

In most cases, you'll never need to use the `GetMessage()` or the `PeekMessage()` functions because CLR handles these requirements for you. However, these functions do come in handy for special messages (see the `RegisterWindowMessage()` section that follows) or within wrapper DLLs. What's most important is to understand the mechanism used to retrieve the messages once they arrive in the queue.

The `GetMessage()` function requires four inputs. The *lpMsg* argument is the most important because it contains a pointer to the `Msg` data structure used to hold the message information. When the call returns, the `Msg` data structure contains the information needed to process the message. The *hWnd* argument contains a handle to a window. However, you can set *hWnd* to `NULL` if you want to retrieve a given message for all windows associated with the current process. The *wMsgFilterMin* and *wMsgFilterMax* arguments contain a range of messages that you want to retrieve based on the value for each message (see the C header files for a complete list—the various examples in the chapter have already shown you the values of specific messages). If you want to retrieve a single message, then you set the *wMsgFilterMin* and *wMsgFilterMax* arguments to the same value. There are also predefined ranges values such as `WM_MOUSEFIRST` and `WM_MOUSELAST` that obtain specific input values.

The `PeekMessage()` function requires all of the arguments used by the `GetMessage()` function. You also need to provide a *wRemoveMsg* argument value. A value of `PM_REMOVE` will remove the message from the queue, while a value of `PM_NOREMOVE` will keep the message on the queue. Given the reason for using `PeekMessage()`, you'll probably use `PM_NOREMOVE` in most cases.

## RegisterWindowMessage()

You'd think that with all of the messages that Windows supports natively, you'd never need to create a message of your own. Actually, applications commonly create custom messages for intra-application communication. Sometimes an application will spawn other processes and need to communicate with those processes using a special set of messages. Because the messages are sent publicly with `SendMessage()` or `PostMessage()`, Windows needs to know about them and you need to provide a unique name for them. The purpose of the `RegisterWindowMessage()` function is to register a unique name for your custom message. All you need to supply is a string containing the message name.

## Creating a Windows Message Handler Example

This chapter already contains several examples that show how to send a message to Windows. Given an application need, you can send a request to Windows to answer that need. In fact, you can affect the operation of all of the applications running under Windows in some circumstances (as was shown with the `MinimizeAll` example). However, there are times when you want to create an environment where Windows can send a message to your application. Of course, this already happens all the time when users click buttons and enter text, but you might have some special message that you want Windows to send to your application that the .NET Framework doesn't handle by default.

The example in this section shows how to create a message handler that will react when Windows sends a specific message. To do this, we have to override the default .NET functionality for the Windows message pump, create an event that the message pump will fire when it receives the message in question, and create an event handler that does something when it receives an event notification. The following example is a prototype of sorts for handling all kinds of Windows messages. You'll see more advanced examples of this technique in Chapter 9 when we tackle advanced Windows XP features such as Fast User Switching. You'll find the code for this example in the `\Chapter 04\C#\ReceiveMessage` and `\Chapter 04\VB\ReceiveMessage` folders of the CD.

## Creating the Event

The event portion of the code generates an event when requested. It will send the event to any number of handlers—all of which must register to receive the event notification. The event portion of the code doesn't do anything with the event notification; it merely reacts to the event and generates the notification. This is an extremely important distinction to consider. Listing 4.3 shows the event code for this example.

**Listing 4.3****The Event Code for a Message Handler**

```
// Create an event for the message handler to fire. We also
// have to handle this event or nothing will happen.
public delegate void DoSDCheck(object sender, System.EventArgs e);
public static event DoSDCheck ThisSDCheck;

// Provide a means for firing the event.
public static void Fire_ThisSDCheck(object sender, System.EventArgs e)
{
    // If there is an event handler, call it.
    if (ThisSDCheck != null)
        ThisSDCheck(sender, e);
}
```

---

As you can see, you need a delegate to define the appearance of the event handler. `DoSDCheck()` isn't an event handler; it merely acts as a prototype for the event handler. The event is an instance of the delegate. You must make the event static or no one will be able to call it.

Once you have an event defined, you need a way to fire it. Microsoft doesn't define the name of the method for firing an event in any concrete terms, but standard practice is to preface the event name with the word "Fire" followed by an underscore, so the name of this method is `Fire_ThisSDCheck()`. Firing an event can require a lot of work; but generally all you need to do is verify that the event has at least one handler, and then call the event. This step will call every assigned event handler in turn to process the event notification.

## Creating the Windows Message Loop Override

The most important tip you can remember about processing messages is that the .NET Framework will only handle the messages that applications commonly use. If you need any other functionality in your application, then you need to add it. Common functionality includes messages associated with the mouse and the keyboard—it doesn't include messages associated with a shutdown.

**TIP**

Sometimes the Platform SDK documentation is simply wrong. For instance, the documentation for the `WM_QUERYENDSESSION` message used in this example tells you that it's sent in response to an `ExitWindows()` function call. Unfortunately, Windows XP doesn't support the `ExitWindows()` function, so there's no hope of making this function work properly given the documentation. You need to use the `ExitWindowsEx()` function instead. The best way to find this information is to use the Dependency Walker to view `User32.DLL` and see if it supports the `ExitWindows()` function. The answer becomes obvious during the few seconds it takes to check the DLL.

With this in mind, you have to rewrite the message pump to do something with the messages that you want to handle. This means overriding the default message pump, and then calling the base message pump to handle any messages that your code doesn't handle. The two-step process is important. If you don't call the base function, then any messages your code doesn't handle will go unanswered. Of course, you can always use this technique to force an application to handle just a few messages and ignore everything else—a somewhat dangerous proposition unless you know exactly what you're doing. Listing 4.4 shows the message pump override required for this example.

**Listing 4.4      Always Override the Message Pump to Handle Custom Messages**

```
// We need to know which message to monitor.
public const Int32 WM_QUERYENDSESSION = 0x0011;
public const Int32 WM_ENDSESSION = 0x0016;

protected override void WndProc(ref Message ThisMsg)
{
    // See which message Windows has passed.
    if ((ThisMsg.Msg == WM_QUERYENDSESSION) ||
        (ThisMsg.Msg == WM_ENDSESSION))
    {
        // Fire the event.
        Fire_ThisSDCheck(this, null);

        // No more processing needed.
        return;
    }

    // If this isn't a session end message, then pass the
    // data onto the base WndProc() method. You must do this
    // or your application won't do anything.
    base.WndProc(ref ThisMsg);
}
```

The code for the message pump is relatively straightforward. All you need to do is check for the session ending messages, and then fire the event. Notice that we return from this function without providing a positive response to Windows. This omission enables the application to cancel the shutdown. If you want to allow the system to shut down, you must set the *ThisMsg.Result* value to **true**.

## Creating the Event Handler

The event handler for this example doesn't do much—it displays a message box saying it received a notification. However, it's important to realize that the message handler could do anything within reason. Windows sets a time limit for responding to a shutdown message. If your event handler is code heavy, your application won't respond in time and Windows will try to shut it down manually. Listing 4.5 shows the event handler for this example.



**Listing 4.5**     **The Event Handler for the Example is Simple and Fast**

```
public frmMain()
{
    // Required for Windows Form Designer support
    InitializeComponent();

    // Add an event handler for the shutdown check.
    ThisSDCheck += new DoSDCheck(OnShutDownCheck);
}

// Create an event handler for the shutdown event.
private void OnShutDownCheck(object sender, System.EventArgs e)
{
    // Display a message showing that we received the message.
    MessageBox.Show("Windows sent an end session message",
                    "End Session Message",
                    MessageBoxButtons.OK,
                    MessageBoxIcon.Information);
}
```

---

Notice that you must register the event handler. Otherwise, it won't receive event notifications. In this case, the example registers the event handler in the constructor, which is a good place for the registration for most applications. If an event handler is important enough to monitor messages from Windows, you'll want to register it during the application startup process.

## Demonstrating the Windows Message Handler

In older versions of Windows you simply told the operating system that you wanted to shut down, and that was the end of the process. Newer versions of Windows require a little more

information and Windows XP makes it downright impossible to shut down unless you have a good reason. For this reason, the code for initiating a Windows shutdown is a bit long. Listing 4.6 provides you with the essentials.



#### **Listing 4.6 Using the ExitWindowsEx() Function to Shut Windows Down**

```
// Used to send a message that starts the screen saver.
[DllImport("User32.DLL")]
public static extern int ExitWindowsEx(UInt32 uFlags,
                                       UInt32 dwReason);

// A list of flags that determine how the system is shut down.
public enum ShutdownFlag
{
    EWX_LOGOFF           = 0,
    EWX_SHUTDOWN         = 0x00000001,
    EWX_REBOOT           = 0x00000002,
    EWX_FORCE             = 0x00000004,
    EWX_POWEROFF         = 0x00000008,
    EWX_FORCEIFHUNG      = 0x00000010
}

// A list of major reasons to shut the system down.
public enum ReasonMajor
{
    SHTDN_REASON_MAJOR_OTHER           = 0x00000000,
    SHTDN_REASON_MAJOR_NONE           = 0x00000000,
    SHTDN_REASON_MAJOR_HARDWARE       = 0x00010000,
    SHTDN_REASON_MAJOR_OPERATINGSYSTEM = 0x00020000,
    SHTDN_REASON_MAJOR_SOFTWARE       = 0x00030000,
    SHTDN_REASON_MAJOR_APPLICATION    = 0x00040000,
    SHTDN_REASON_MAJOR_SYSTEM         = 0x00050000,
    SHTDN_REASON_MAJOR_POWER          = 0x00060000
}

// A list of minor reasons to shut the system down. Combine
// these reasons with the major reasons to provide better
// information to the system.
public enum ReasonMinor
{
    SHTDN_REASON_MINOR_OTHER           = 0x00000000,
    SHTDN_REASON_MINOR_NONE           = 0x000000ff,
    SHTDN_REASON_MINOR_MAINTENANCE     = 0x00000001,
    SHTDN_REASON_MINOR_INSTALLATION    = 0x00000002,
    SHTDN_REASON_MINOR_UPGRADE         = 0x00000003,
    SHTDN_REASON_MINOR_RECONFIG        = 0x00000004,
    SHTDN_REASON_MINOR_HUNG            = 0x00000005,
    SHTDN_REASON_MINOR_UNSTABLE        = 0x00000006,
    SHTDN_REASON_MINOR_DISK            = 0x00000007,
    SHTDN_REASON_MINOR_PROCESSOR       = 0x00000008,
```



```

        SHTDN_REASON_MINOR_NETWORKCARD          = 0x00000009,
        SHTDN_REASON_MINOR_POWER_SUPPLY         = 0x0000000a,
        SHTDN_REASON_MINOR_CORDUNPLUGGED        = 0x0000000b,
        SHTDN_REASON_MINOR_ENVIRONMENT          = 0x0000000c,
        SHTDN_REASON_MINOR_HARDWARE_DRIVER       = 0x0000000d,
        SHTDN_REASON_MINOR_OTHERDRIVER          = 0x0000000e,
        SHTDN_REASON_MINOR_BLUESCREEN           = 0x0000000F,
        SHTDN_REASON_UNKNOWN                    = SHTDN_REASON_MINOR_NONE
    }

    // A list of reason flags that provide additional information about the
    // cause of shutdown. Combine these flags with the major and minor reason
    // values.
    public enum ReasonFlag : uint
    {
        SHTDN_REASON_FLAG_USER_DEFINED           = 0x40000000,
        SHTDN_REASON_FLAG_PLANNED               = 0x80000000
    }

    private void btnTest_Click(object sender, System.EventArgs e)
    {
        // Exit Windows.
        ExitWindowsEx((UInt32)ShutdownFlag.EWX_LOGOFF,
            (UInt32)ReasonMajor.SHTDN_REASON_MAJOR_APPLICATION &
            (UInt32)ReasonMinor.SHTDN_REASON_MINOR_MAINTENANCE &
            (UInt32)ReasonFlag.SHTDN_REASON_FLAG_PLANNED);
    }

```

---

There are a lot of predefined reasons for shutting the system down and you should choose one of them within your application. Generally, you'll choose the appropriate *ShutdownFlag* value for the first argument. Notice there are options for logging off, performing a normal reboot, and forcing a shutdown for a hung application. This last option should be used with care, but it's a valuable option if an application detects that it has frozen and the system is in an unstable state. (Of course, recovering from the condition is even better.)

I decided to split the second argument into three enumerations because each enumeration performs a different task. You should always include a *ReasonMajor* value as part of the shutdown. The *ReasonMinor* value further defines the reason for the shutdown, but isn't essential. Finally, you can pass a *ReasonFlag* value if one of the values happens to meet your needs.

## Developing for Thread Safety

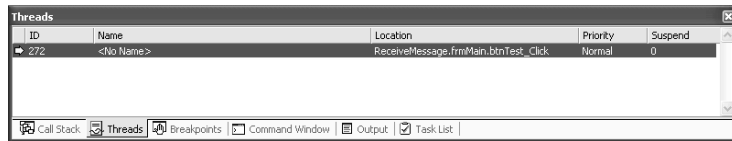
You might think that all of the convoluted code in this example could be straightened out and made simpler. The fact is that the technique shown in this example becomes more important as the complexity of your code increases. The moment you introduce a second thread into the

application, the need for all of the convoluted code becomes essential. Using events as we have here keeps the message handling in the main thread.

One of the Visual Studio IDE windows that you need to look at is the Threads window. Unfortunately, the Visual Studio IDE hides this window by default and most developers don't find it because it's hidden on the Debug menu instead of the View menu. To display the Threads window, use the Debug > Windows > Threads command. Figure 4.8 shows an example of the Threads window for the current application.

**FIGURE 4.8:**

The Threads window can be helpful in diagnosing problems with a Win32 API message handler.



Any code that changes the appearance of a Windows Form must execute from the main thread of the application. This is why you want to use an event handler for your message handling code. Using an event handler means that no matter which thread intercepts the message you want to process, the main thread will perform the actual processing.

## Where Do You Go from Here?

This chapter has demonstrated various uses for Windows messages in managed applications. Like unmanaged Windows applications, managed applications use messaging to communicate between applications and the operating system. Knowing which Windows messages the .NET Framework supports natively can help you determine when you need to create a handler for non-standard messages.

We have discussed the correlation between some .NET Framework event handlers and the Win32 API messages. Create a small test application and use Spy++ to verify the messages that it responds to. Add objects such as menus to see the effect on the output messages. Remember to limit the message selections in Spy++ so that you can actually see the messages of interest—some messages (especially those for mouse handling) appear with such regularity that it's hard to see the messages that appear only when specific events occur.

Make sure you try out all of the examples on the CD. There are places in the chapter where I mention an example, but don't go completely through the code because most of it has appeared in other chapters. It's still important to check the example out because you'll learn techniques for working with messages by using them. Especially important are some of the system commands that aren't handled very well by the .NET Framework.

Now that you know about messages, it's time to look at the last generic feature for Win32 API programming—the callback function. Chapter 5 tells you how Windows uses callback functions for various tasks and when you'll need to use them for your Win32 API call. Callback functions are important because they provide a mechanism for Windows to interact with an application. Essentially, the application makes a request and Windows answers it through the callback function. This asynchronous handling of application requests enables Windows to run more efficiently, but does add to the developer's workload.